DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems

Chang Joo Lee *Veynu Narasiman* *Eiman Ebrahimi* *Onur Mutlu‡* *Yale N. Patt*

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890

TR-HPS-2010-002
April 2010

This page is intentionally left blank.

DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems

Abstract

Read and write requests from a processor contend for the main memory data bus. System performance depends heavily on when read requests are serviced since they are required for an application's forward progress whereas writes do not need to be performed immediately. However, writes eventually have to be written to memory because the storage required to buffer them on-chip is limited.

In modern high bandwidth DDR (Double Data Rate)-based memory systems write requests significantly interfere with the servicing of read requests by delaying the more critical read requests and by causing the memory bus to become idle when switching between the servicing of a write and read request. This interference significantly degrades overall system performance. We call this phenomenon write-caused interference. To reduce write-caused interference, this paper proposes a new last-level cache writeback policy, called DRAM-aware writeback. The key idea of the proposed technique is to aggressively send out writeback requests that are expected to hit in DRAM row buffers before they would normally be evicted by the last-level cache replacement policy and have the DRAM controller service as many writes as possible together. Doing so not only reduces the amount of time to service writes by improving their row buffer locality but also reduces the idle bus cycles wasted due to switching between the servicing of a write and a read request.

DRAM-aware writeback improves system performance by 7.1% and 12.8% on single and 4-core systems respectively. The performance benefits of the mechanism increases in systems with prefetching since such systems have higher contention between reads and writes in the DRAM system.

1. Introduction

Read and write requests from the processor contend for the main memory data bus. In general, read requests (i.e., miss requests from the last-level cache) are critical for system performance since they are required for an application's progress whereas writes (i.e., writeback requests from the last-level cache) do not need to be performed immediately. In modern DDR (Double Data Rate)-based memory systems, write requests significantly interfere with the servicing of read requests, which can degrade overall system performance by delaying the more critical read requests. We call this phenomenon *write-caused interference*. There are two major sources of performance penalty when a write request is serviced instead of a read request. First, the critical read request is delayed for the duration of the service latency of the write request. Second, even after the write is serviced fully, the read cannot be started because the DDR DRAM protocol requires additional timing constraints to be satisfied which causes idle cycles on the DRAM data bus in which no data transfer can be done.

The two most important of these timing constraints are write-to-read (^tWTR) and write recovery (write-to-precharge, ^tWR) latencies as specified in the current JEDEC DDR DRAM standard [3]. These timing constraints in addition to other main access latencies such as precharge, activate and column address strobe latencies (^tRP,^{t} RCD, and CL/CWL) dictate the number of cycles in which the DRAM data bus should remain idle after a write, before a read can be performed. In a state-of-the-art DDR3 DRAM system ^tWTR and ^tWR latencies are 7.5 and 15 ns [9], which translates to 30 and 60 processor cycle delays assuming a processor clock frequency of 4 GHz. Both latencies increase in terms of number of DRAM clock cycles as the operating frequency of the DRAM chip increases [16, 3] as do other main access latencies. The end result is that high penalties caused by write requests will become even larger in terms of number of cycles because the operating frequency of future DRAM chips will continue to increase to maintain high peak bandwidth.

A *write buffer* in the main memory system can mitigate this problem. A write buffer holds write requests on-chip until they are sent to the memory system according to the write buffer management policy. While write requests are held by the write buffer, read requests from the processor can be serviced by DRAM without interference from write requests. As a result, memory service time for reads that are required by the application can be reduced. As the write buffer size increases, write-caused interference in the memory system decreases. For example, an infinite write buffer can keep all write requests on-chip, thereby completely

removing write-caused interference. However, a very large write buffer is not attractive since it requires high hardware cost and high design complexity (especially to enable forwarding of data to matching read requests) and leads to inefficient utilization of on-chip hardware/power budget. In fact, a write buffer essentially acts as another level of cache (holding only written-back cache lines) between the last-level cache and the main memory system.

To motivate the performance impact of write-caused interference, Figure 1 shows performance on a single-core system (with no prefetching) that employs a state-of-the-art DDR3-1600 DRAM system (12.8 GB/s peak bandwidth) [9] and a First Ready-First Come First Served (FR-FCFS) DRAM controller [15]. We evaluate three write request management policies: 1) a 64-entry write buffer with a management policy similar to previous proposals [6, 12, 17] which exposes writes (i.e., makes them visible) to the DRAM controller only when there is no pending read request or when the write buffer is full, and stops exposing writes when a read request arrives or when the write buffer is not full anymore (*service_at_no_read*), 2) a 64-entry write buffer with a policy that exposes all writes only when the write buffer is full and continues to expose all writes until the write buffer becomes empty (*drain_when_full*), and 3) ideally eliminating all writes assuming that there is no correctness issue (no_write). Ideally eliminating all writes removes all write-caused interference and therefore shows the upper bound on performance that can be obtained by handling write-caused interference intelligently.[1]

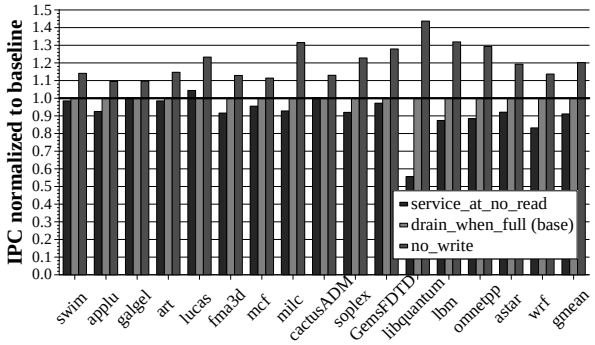


Figure 1. Potential performance of intelligently handling write-caused interference in the DRAM system

We make two main observations. First, *service_at_no_read* usually performs worse than servicing writes when the write buffer is full. This is because when a read arrives at the DRAM controller very soon after a write is serviced, a significant amount of write-caused penalty delays that read. This happens to all the benchmarks except for *lucas* where there are long enough periods to satisfy the large write-caused penalties during which reads are not generated. Servicing writes opportunistically when there are no reads degrades performance due to two reasons: 1) it incurs the costly write-to-read and read-to-write switching penalties, thereby wasting DRAM bandwidth, 2) it does not exploit row buffer locality when servicing write requests since writes that go to the same row are serviced far apart from each other in time. In contrast, *drain_when_full* improves performance by 9.8% compared to *service_at_no_read* on average because it 1) delays service of writes as much as possible, 2) services all writes once it starts servicing one write, thereby amortizing write-to-read switching penalties across multiple writes by incurring them only once for an entire write-buffer worth of writes, and 3) increases the possibility of having more writes to the same DRAM row address or higher *row buffer locality* in the write buffer that is exploited by the DRAM controller for better DRAM throughput. Second, even though *drain_when_full* significantly improves performance compared to the best existing write buffer management policies, there is still large potential performance improvement (20.2% compared to *drain_when_full*) that can be achieved by further reducing write-caused interference, as shown by the rightmost set of bars.

[1]We chose 16benchmarks among all SPEC2000/2006 CPU benchmarks that have at least 10% IPC (retired instruction per cycle) performance improvement compared to *drain_when_full* when all writes are ideally removed. The performance numbers shown in Figure 1 are normalized to *drain_when_full*. Section 5 describes our experimental methodology in detail.

As shown above, the impact of write-caused interference on an application's performance is significant even with a decently-sized (i.e., 64-entry) write buffer and a good write buffer policy. This is because a size-limited write buffer or a write buffer management policy cannot completely remove write-caused interference since 1) writes eventually have to be written back to DRAM whenever the write buffer is full and 2) servicing all writes in the write buffer still consumes a significant amount of time. To overcome this problem, we propose a new last-level cache writeback policy called *DRAM-aware writeback* that aims to maximize DRAM throughput for write requests in order to minimize write-caused interference. The basic idea is to send out writebacks that are expected to hit in DRAM row buffers before they would normally be evicted by the last-level cache replacement policy. This allows higher row buffer locality to be exposed in an existing write buffer which the DRAM controller can take advantage of. Once the write buffer becomes full, the DRAM controller services writes quickly (since they would hit in row buffers) until all writes in the write buffer are serviced. Our mechanism is able to continue to send more writes to the write buffer while the DRAM controller is servicing writes. This allows the DRAM controller to service more writes once it starts servicing writes thereby resulting in less frequent write-to-read switching later.

Our evaluations show that the proposed mechanism improves system performance significantly by managing DRAM write-caused interference, which in turn increases DRAM bus utilization. The DRAM-aware writeback mechanism improves the performance of 18 memory intensive SPEC CPU 2000/2006 benchmarks by 7.1% on a single-core processor compared to the best write buffer policy among policies we evaluated. It also improves system performance (i.e. harmonic speedup) of 30 multiprogrammed workloads by 12.8% on a 4-core CMP. We show that our mechanism is simple to implement and low-cost.

Contributions To our knowledge, this is the first paper that addresses the write-caused interference problem in state-of-the-art DDR DRAM systems. We make the following contributions:

1. We show that write-caused interference in DRAM is and will continue to be a significant performance bottleneck in modern and future processors.

2. We show that a simple write buffer management policy that services all writes only when the write buffer is full outperforms previously proposed policies by reducing DRAM write-to-read switching penalties.

3. We propose a new writeback policy for the last-level cache that takes advantage of the best write buffer management policy and reduces the service time of DRAM writes by exploiting DRAM row buffer locality. The proposed writeback mechanism improves DRAM throughput for both reads and writes by reducing write-caused interference.

4. We evaluate our techniques for various configurations on single-core and CMP systems, and show that they significantly improve system performance on a wide variety of system configurations.

2. Background

2.1. Write-Caused Interference in DRAM systems

Write-caused interference in DRAM comes from read-to-write, write-to-read, and write-to-precharge latency penalties. Read-to-write and write-to-read latencies specify the minimum idle latencies on the data bus between a read and a write regardless of what DRAM banks they belong to. In contrast, write-to-precharge specifies the minimum latency between a write command and a subsequent precharge command to the same bank. We first describe read-to-write and write-to-read latencies.

2.1.1. Write-to-Read and Read-to-Write Penalties and How to Reduce Them Read-to-write latency is the minimum latency from a read data burst to a write data burst. This latency is required to change the data bus pins' state from read state to write state. Therefore, during this latency the bus has to be idle. In DDR3 DRAM systems, the read-to-write latency is two DRAM clock cycles. Write-to-read (^tWTR) latency is the minimum latency from a write burst to a subsequent read command. In addition to the

time required for the bus state change from write to read, this latency also includes the time required to guarantee that written data can be safely written to the row buffer (i.e., sense amplifier) such that a possible subsequent read to the same row can be performed correctly. Therefore ${}^tWTR$ is much larger (e.g., 6 DRAM clock cycles for DDR3-1600) than read-to-write latency and introduces more DRAM data bus idle cycles. Note that both of the latencies must be satisfied regardles of whether the read and the write access the same bank or different banks.

We demonstrate the implications of these penalties on DRAM throughput and overall performance with an example in Figure 2. Figure 2(a) shows the state of the DRAM read and write buffers. For brevity, we assume that each buffer has only two entries in this example. All the read requests in the DRAM read buffer are always exposed to the DRAM controller for scheduling whereas the writes are exposed based on the write buffer management policy. There is one read (Read A, a read request to row buffer A) and one write (Write B) in the read and write buffers respectively. At time t1, another read (Read C) and a write (Write D) come from the processor. We assume that each request goes to a different bank and that all requests match the open row buffer in their corresponding DRAM banks (all requests are row-hits).

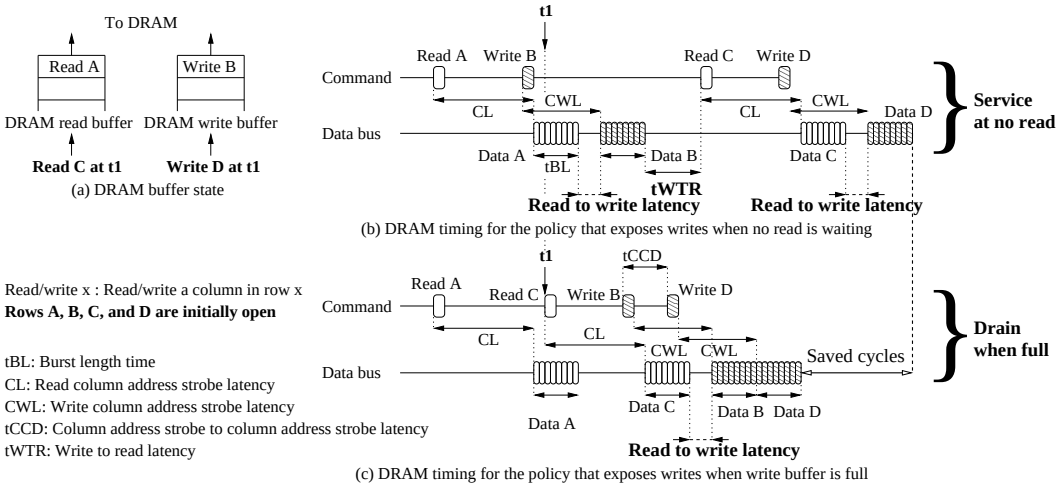


Figure 2. Effect of read-to-write and write-to-read penalties based on write buffer management policies

Figure 2(b) shows the DRAM timing diagram for the policy which exposes writes to the DRAM controller only when there is no pending read request or when the write buffer is full and stops exposing writes when a read request comes in or when the write buffer is not full anymore (the *service_at_no_read* policy in Section 1). Since no read is pending in the DRAM read buffer after Read A is scheduled, this policy schedules Write B from the write buffer. Subsequently Read C and Write D are scheduled.

Each command (e.g., read, write, or precharge) takes a DRAM bus cycle and every data transfer is done in burst mode (BL, Burst length of 8 in the figure) at twice the rate of the clock (i.e., double data rate, 4 DRAM clock cycles for BL = 8). Two observations can be made from Figure 2(b) which will demonstrate the problems caused by write-to-read and read-to-write latencies. First, the command for Write B after Read A must satisfy read-to-write latency; it has to be scheduled by the DRAM controller at least $CL +^t BL + 2 - CWL$ [3] DRAM clock cycles after the read command is scheduled such that the write burst can be on the bus two DRAM cycles after the read burst [2]. Second, Read C after Write B must satisfy ${}^tWTR$. The command for Read C can only be scheduled ${}^tWTR$ cycles after the data burst for Write B is completed. In contrast to read-to-write latency, the data bus must be idle for ${}^tWTR + CL$ cycles since the subsequent read command cannot be scheduled for ${}^tWTR$ cycles. The last write is scheduled after read-to-write latency is satisfied as shown.

[2]We assume that the additive latency (AL) is 0 in this study. If AL is considered, the subsequent write command can be scheduled $CL + AL +^t CCD + 2 - (CWL + AL)$ cycles after the read, where ${}^tCCD$ is the minimum column strobe to column strobe latency). To maximize bandwidth we set up ${}^tBL$ to eight, therefore ${}^tCCD$ is equal to (tBL) [3].

This policy results in many idle cycles (i.e., poor DRAM throughput) on the data bus. This is because it sends writes as soon as there are no pending reads which is problematic when a subsequent read arrives immediately after the write is scheduled to DRAM. The penalties introduced by the write cause a significant amount of interference and therefore increase both the read's and write's service time. This is the main reason why this policy does not perform well as shown in Figure 1.

On the other hand, if the write buffer policy that exposes all writes only when the write buffer is full and continues to expose all writes until the write buffer becomes empty (*drain_when_full*) is used, Reads A and C are serviced first (Write B is not serviced immediately after Read A since the write buffer is not full) and then Writes B and D are serviced. Figure 2(c) shows the DRAM timing diagram for this policy. Read C can be scheduled once the DRAM controller sees it since there is no unsatisfied timing constraint for Read C. Then Write B can be scheduled $CL +^t BL + 2 - CWL$ cycles after the command for Read A is scheduled. Note that the command for Write D can be scheduled very soon (more precisely, ^tCCD cycles after the command for Write B) since DDR3 DRAM chips support back-to-back data bursts for writes (as well as for reads) by overlapping column address strobe latencies (CL or CWL).

This policy results in better DRAM service time for the four requests compared to the policy shown in Figure 2(b). Since buffering writes in the DRAM write buffer and servicing all of them together when the buffer gets full reduces the large read-to-write and write-to-read latency penalties, DRAM throughput increases. Also note that by delaying writes as much as possible, reads that are more critical to an application's progress can be serviced quickly thereby improving performance. This is the main reason this policy outperforms the policy of Figure 2(b) as shown in Figure 1. We found that this policy is the best among the previously proposed write buffer policies we evaluated. We use this policy as our baseline write buffer policy.

2.1.2. Write-to-Precharge Penalty In the previous example we assumed that all rows for the four requests are open in the row buffers of their corresponding DRAM banks. Write-to-precharge latency (write recovery time, ^tWR) comes into play when a subsequent precharge command (for either a read or a write) is scheduled to open a different row after a write data transfer in a bank. This write-to-precharge latency specifies the minimum latency from a write data burst to a precharge command in the same DRAM bank. This latency is very large (12 DRAM clock cycles for DDR3-1600) because the written data in the bank must be written back to the corresponding DRAM row through the row buffer before precharging the DRAM bank. This needs to be done to avoid the loss of modified data.

Figure 3 illustrates write-to-precharge penalty in a DRAM bank. Write A and Read B access different rows in the same bank. Therefore a precharge command is required to be sent to DRAM to open the row for Read B. Subsequent to the scheduling of Write A, the precharge command must wait until write-to-precharge latency is satisfied before it can be scheduled. Note that this penalty must be satisfied regardless of whether the subsequent precharge is for a read or write.

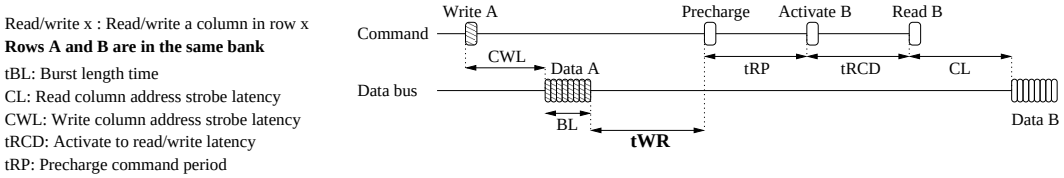


Figure 3. Write-to-precharge latency (^tWR)

This write-to-precharge latency affects DRAM throughput mainly in two ways. First, when reads and writes to different rows (i.e., row-conflict) are serviced alternatively, the total amount of write-to-precharge penalty becomes very large. For example, servicing Write A (write to row A), Read B, Write A, and Read B in a bank will result in poor service time by introducing large penalties (3 row-conflict and 2 write-to-precharge latencies). This can be mitigated by the write buffer policy that exposes all writes

to the DRAM controller only when the write buffer is full. By doing so, first the two writes to row A are serviced and then the two reads to row B are serviced (resulting in 1 row-conflict and 1 write-to-precharge latency).

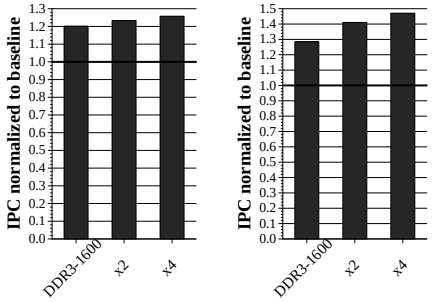
Second, since the write-to-precharge latency must be satisfied even for a subsequent precharge for a write, row-conflicts among writes degrade DRAM throughput for writes. For example, Write B after Write A must still satisfy this write-to-precharge penalty before the precharge to open row B can be scheduled. This problem cannot be solved by write buffer policies. If writes in the write buffer access different rows in the same bank, the total amount of write-to-precharge penalty becomes very large. This degrades DRAM throughput for writes even with the write buffer policy that exposes writes only when the write buffer is full. This eventually results in delaying service of reads thereby degrading application performance.

3. Motivation

3.1. Performance Impact of Write-caused Interference in the Future

We expect that write-caused interference will continually increase in terms of number of clock cycles as the operating frequency of the DRAM chip increases to maintain high peak bandwidth. The write-to-read penalty which guarantees that modified data is written to the row buffer correctly (sense amplifier) will not be easily reduced in absolute time similar to other access latencies such as precharge period (^tRP) and column address strobe latency (CL/CWL). This is especially true for the write-to-precharge latency which guarantees modified data will be completely written back to the memory rows before a new precharge. This latency cannot easily be reduced because reducing access latency to the memory cell core is very difficult [16, 3]. We believe this will be true for any future memory technology (not limited to DRAM technology) that supports high peak bandwidth. This means that write-caused interference will continue to be a performance bottleneck in the future.

Figure 4 shows the performance improvement of the ideal writeback policy across future high bandwidth memory systems. We assume that the DRAM operating frequency continue to increase in the future. Since the future memory specifications are unknown, we speculatively scaled the number of clock cycles for all DDR3-1600 performance-related latencies that cannot be easily reduced (e.g., ^tWTR, ^tWR, ^tRP, ^tRCD, CL, etc) in absolute time. For example, x2 of DDR3-1600 indicates a DDR system that maintains twice the DDR3-1600 peak bandwidth (25.6GB/s = 2 x 12.8GB/s). We also assume that the DRAM frequency increases as fast as the processor frequency. We show two cases: when no prefetching is employed and when an aggressive stream prefetcher is used in the processor.



(a) No prefetching (b) Prefetching

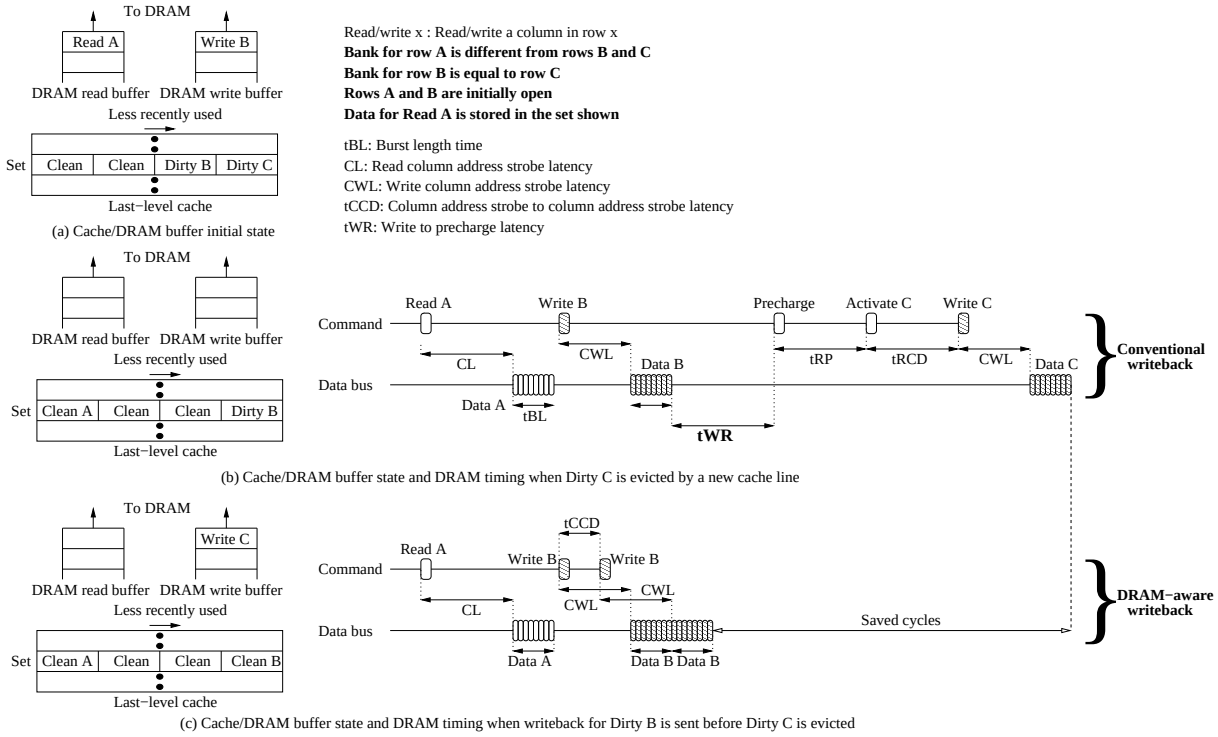
Figure 4. Performance potential as memory operating frequency increases

We make two observations from Figure 4. First, the higher the peak bandwidth, the larger the performance impact of write-caused interference. Second, removing write-caused interference is more critical for systems with prefetching. The performance impact of writes for these systems is much higher due to higher contention between reads and writes (prefetch requests are all reads).

3.2. Last-Level Cache Writeback Policy: A Way to Reduce Write-Caused Interference

As discussed in Section 2.1.2, write-to-precharge penalty cannot be reduced by write buffer policies (such as drain when full). Servicing row-conflict writes in the same bank takes a significant number of cycles. This will delay service of writes in the write buffer and eventually results in delaying service of reads. Service of writes can be done faster if the write buffer has many row-hit writes. Note that the source of DRAM writes is the last-level cache's writebacks which are dirty line evictions in a writeback cache. In contrast to read requests that are required immediately for an application's progress, writes can be scheduled to DRAM more freely. For example, the last-level cache can more aggressively send out writebacks even though no dirty line is evicted by its cache replacement policy to improve service time of writes.

Figure 5 shows an example of an aggressive writeback policy of the last-level cache. Figure 5(a) shows the initial state of the DRAM read/write buffers and a set of the last-level cache. A read (Read A, read to row A) and a write (Write B, write to row B) are waiting to be scheduled to DRAM in the DRAM read and write buffers (two entries for each) respectively. Two dirty lines (Dirty C and Dirty B) are at the least recently used (LRU) positions of the shown set of the last-level cache. For simplicity, we assume that rows B and C are mapped to the same bank, whereas row A is mapped to a different bank. Also row A and row B are open in the row buffers of the respective banks. Read A is about to be scheduled and will be inserted in the shown set of the cache.



(a) Cache/DRAM buffer initial state

(b) Cache/DRAM buffer state and DRAM timing when Dirty C is evicted by a new cache line

(c) Cache/DRAM buffer state and DRAM timing when writeback for Dirty B is sent before Dirty C is evicted

Figure 5. How last-level cache's writeback policies affect DRAM service time

Figure 5(b) shows the resulting cache and buffer states and the DRAM timing when a conventional writeback policy is used in the cache. The LRU line (Dirty C) is evicted by the new line for Read A after read A is serviced by DRAM. Therefore a write is generated for row C (Write C) and is inserted into the write buffer. The write buffer becomes full since it contains two writes. Subsequently, the baseline write buffer policy (*drain_when_full*) allows the DRAM controller to schedule both writes. Write B is scheduled first since it is a row-hit and write C is serviced next. Because Write C accesses a different row from Write B, precharging is required to open row C. Since a write was serviced before, write-to-precharge penalty must be satisfied before the precharge command for C is scheduled. This increases the idle cycles on the DRAM data bus since the write data for Write C must

wait for ${}^tWR +^t RP +^t RCD + CWL$ cycles after the write burst for Write B.

On the other hand, as shown in Figure 5(c), if the writeback for Dirty B in the cache can be sent out before Dirty C is evicted due to the new line for Read A, the write buffer will contain two writes to the same row. The two writes to row B are serviced back-to-back thereby resulting in significant reduction in DRAM service time. This example illustrates that a writeback policy which can send out writeback requests that will access the same row as other writes can improve service time for writes. This is because write-to-precharge, precharge, and activate latencies (${}^tWR +^t RP +^t RCD$) that would have been applied to a subsequent row-conflict write can be replaced by a row-hit write. Note that two writes to the same row can be even faster since DDR3 DRAM chips support back-to-back data bursts for writes and this is why the aggressive writeback policy's "Saved cycles" is ${}^tWR+{}^tRP+{}^tRCD+CWL$ in Figure 5.

Note that due to this aggressive writeback, the state of the cache and DRAM read/write buffer differs in the case of the new writeback policy of Figure 5(c) compared to the conventional policy. In the case of the conventional policy, Dirty B stays in the cache and no write is left in the write buffer whereas in the writeback policy of Figure 5(c), a clean (non-dirty) copy for row B stays in the cache and Write C remains in the write buffer. Nonetheless, the aggressive writeback policy can still outperform the conventional writeback, because 1) a clean copy of B does not need to be written back to DRAM unless it is rewritten by a dirty line eviction from the lower-level cache and 2) it may find more dirty lines to row C (the same row as Write C in the write buffer) in the cache and send out the writebacks for them to the write buffer so that those row-hit writes to row C can be serviced fast. The reduced DRAM service time turns into higher performance since the DRAM controller quickly switches to service reads.

4. Mechanism: DRAM-Aware Writeback

Our mechanism, DRAM-aware writeback, aims to maximize the DRAM throughput for write requests in order to minimize write-caused interference. It monitors dirty cache lines (writebacks) that are evicted from the last-level cache and tries to find other dirty cache lines that are mapped to the same row as the evicted line. When found, the mechanism aggressively sends writebacks for those dirty cache lines to DRAM. The *drain_when_full* write buffer policy allows writes to be seen by the DRAM controller when the write buffer is full thereby allowing the DRAM controller to exploit row buffer locality of writes. Aggressively sending writebacks selectively cleans cache lines which can be written back quickly due to the DRAM's open row buffers.

The mechanism consists of a writeback monitor unit and state machines in each last-level cache bank as shown in Figure 6. The writeback monitor unit monitors evicted cache lines from all cache banks until it finds one dirty cache line being evicted. It then records the row address of the cache line in each cache bank's state machine. Once a write's row address is recorded, the state machines start sending out writebacks for dirty lines whose row address is the same as the recorded row address (row-hit dirty lines). To find row-hit dirty cache lines, each state machine shares the port of its cache bank with the demand cache accesses from the lower-level cache. Since the demand accesses are more critical to performance, they are prioritized over the state machine's accesses. Once a row-hit dirty line is found, the line's writeback is sent out through the conventional writeback ports regardless of the LRU position of the cache line. Because the cache lines which are written back in this manner may be reused later, the cache lines stay in the cache and only have their dirty bit reset (they become non-dirty or clean). The state machine in each core keeps sending row-hit writebacks until all possible sets that may include cache lines whose row address is the same as the recorded row address have been checked. When all state machines in the banks finish searching, the writeback monitor unit starts observing the writebacks coming out of the cache to start another set of DRAM-aware writebacks.

The DRAM-aware writeback technique leverages the benefits of the write buffer and the baseline write buffer management policy (*drain_when_full*). Our DRAM-aware writeback technique can send more row-hit writebacks than the number of write buffer entries

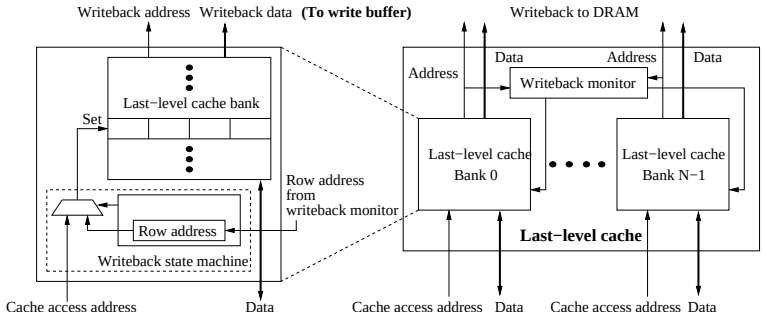


Figure 6. Writeback mechanism in last-level cache

within a very short time. In fact, a single dirty line eviction can trigger our mechanism to send up to row_size/cache_line_size writebacks. Once the write buffer becomes full, all state machines stall and delay the current searching. At the same time, the underlying *drain_when_full* write buffer management policy starts exposing the writes since the write buffer is full. As the DRAM controller services writes, free write buffer entries become available for new writebacks. The state machine resumes searching and sending row-hit writes to the write buffer. Because the *drain_when_full* policy keeps exposing writes until the write buffer becomes empty, all possible row-hit writebacks for a row can be serviced quickly by the DRAM controller since they are all row-hits. In this way, our mechanism can effectively enable more writes to be serviced quickly, which in turn reduces the number of write buffer drains over the entire run of an application. This results in fewer write-to-read switching penalties which improves DRAM throughput and performance.

Note that two conditions should be true for our mechanism to be effective. First, the last-level cache banks should have enough idle cycles for the state machine to look for row-hit writes. If this is true the mechanism would not significantly contend with demand accesses from the lower-level cache for the cache bank and will be able to generate many row-hit writebacks. Second, rewrites to cache lines which our mechanism preemptively writes back to DRAM should not occur too frequently. If writes happen too frequently, the mechanism significantly increases the number of writes to DRAM. Even though row-hit writes can be serviced quickly, the increased writes might increase time spent in servicing writes. We discuss these two issues in the following sections.

4.1. Does Last-Level Cache Have Room for DRAM-Aware Writeback?

Table 1 shows the percent of last-level cache bank idle cycles (averaged over all banks) over the entire run for each of the 16 SPEC2000/20006 benchmarks in a single core system described in Section 5. For all benchmarks, except *art*, cache bank idle time is more than 95%.

Benchmark	swim	applu	galgel	art	lucas	fma3d	mcf	milc	cactusADM	soplex	GemsFDTD	libquantum	lbm	omnetpp	astar	wrf
Idle cycles (%)	0.96	0.97	0.92	0.91	0.98	0.97	0.97	0.97	0.99	0.98	0.97	0.97	0.95	0.98	0.98	0.98

Table 1. Last-level cache bank idle cycles (%) in single core system for 16 SPEC 2000/2006 benchmarks

Table 2 shows the average idle bank cycles of the last-level cache (shared cache for multi-core systems) of the single, 4, and 8-core systems described in Section 5. Even in multi-core systems, the shared last-level cache has many idle cycles. This is because last-level cache accesses are not too frequent compared to lower-level caches, since the lower-level cache and Miss Status Holding Registers (MSHRs) filter out many accesses from the last-level cache. Therefore, we expect contention between demands and our DRAM-aware writeback accesses to be insignificant. We find that prioritizing demands over the accesses for DRAM-aware writeback is enough to reduce the impact of using the cache banks for our mechanism.

	1-core	4-core	8-core
Idle cycles (%)	0.97	0.91	0.89

Table 2. Average last-level cache bank idle cycles (%) in single, 4, and 8-core systems

4.2. Dynamic Optimization for Frequent Rewrites

For applications that exploit temporal locality of the last-level caches, the cache lines which are written back by our aggressive writeback policy may be rewritten by subsequent dirty line evictions of the lower-level cache. These *redirtied* cache lines may come to be written back to DRAM again by the last-level cache's replacement policy or the DRAM-aware writeback policy. This will increase the number of writebacks (i.e., writes to DRAM) which may hurt performance by delaying service of reads due to frequent services for writes.

We mitigate this problem using a simple optimization. We periodically estimate the rewrite rate of cache lines whose writebacks are sent out by the DRAM-aware writeback mechanism. Based on this estimation, our mechanism dynamically adjusts its aggressiveness. For instance, when the rewrite rate is high, the mechanism sends out only row-hit writebacks close to the LRU position. When the rewrite rate is low, the mechanism can send out even row-hit writebacks close to the MRU position. Since the estimation of rewrite rate is periodically done, the DRAM-aware writeback mechanism can adapt to the phase behavior of an application as well. When employing this optimization in the shared cache of a multi-core system, we adapt the mechanism to estimate the rewrite rate for each core (or application).

To implement this, each cache line keeps track of which core it belongs to using core ID bits and also tracks whether the cache line becomes clean (or non-dirty) due to the DRAM-aware writeback mechanism using an additional bit for each line. A counter for each core periodically tracks the total number of the core's writebacks sent out by the DRAM-aware writeback mechanism. Another counter counts the number of the core's rewrites to the clean cache lines whose writebacks were sent early by our mechanism. The rewrite rate for each core for an interval is calculated by dividing the number of rewrites by the total number of writebacks sent out in that interval. The estimated rewrite rate is stored in a register for each core and used to determine how aggressively the mechanism sends writebacks (from LRU or from other positions close to MRU) for the next interval.

We found that our mechanism without this optimization slightly degrades performance for only two applications (*vpr* and *twolf*, both of which are memory non-intensive) out of all 55 SPEC2000/2006 benchmarks by increasing the number of writebacks. Therefore the gain from this optimization is small compared to design effort and hardware cost. We analyze this optimization with experimental results in detail in the results section (Section 6.2).

4.3. Implementation and Hardware Cost

As shown in Figure 6, our DRAM-aware writeback mechanism requires a simple state machine in each last-level cache bank and a monitor unit. Most of the hardware cost is in logic modifications. For example, the comparator structure should be modified to support tag comparison with the row address in each state machine. The only noticeable storage cost is eight bytes per cache bank for storing the row address of the recent writeback. Note that none of the last-level cache structure is on the critical path. As Figure 4.1 shows, the accesses to the last-level cache are not very frequent.

If we implement the optimization in Section 4.2, one additional bit and core ID bits (for multi-core systems) for each cache line are required. Three counters (2 bytes for each) are required to keep track of the number of writebacks sent, the number of rewrites, and the rewrite rate.

4.4. Comparison to Eager Writeback

Eager writeback [6] was proposed to make efficient use of bus idle cycle for writes in a Rambus DRAM system in order to minimize read and write contention. It sends writebacks for dirty LRU lines in a set to the write buffer when the set is accessed. Writes in the write buffer are scheduled when the bus is idle. There are important key differences between eager writeback and our DRAM-aware writeback technique which we discuss below.

First, eager writeback is not aware of DRAM characteristics. We find that simply sending writebacks for dirty LRU cache lines does not work with today's high-frequency DDR DRAM systems because servicing those writes in DRAM is not necessarily completed quickly. For instance, servicing row-conflict writes causes large penalties (write-to-precharge latencies) as shown in Section 3. This eventually significantly delays the service of subsequent reads.

Second, the write-caused penalties of state-of-the-art DDR DRAM systems are too large to send a write only because the data bus is idle or there are no pending reads. To tolerate the long write-caused penalties, there must be no read request arriving at the DRAM system for a long time such that all write-caused timing constraints are satisfied before the subsequent read. However, for memory intensive applications whose working set does not fit in the last-level cache, it is very likely that read requests arrive at the DRAM system before all constraints are satisfied. Therefore subsequent reads suffer large write-to-read penalties.

In contrast, our mechanism does not aim to minimize immediate write-caused interference but targets minimizing the write-caused penalties for the entire run of an application. It allows to stop servicing current reads to service writes. However, once it does, it makes the DRAM controller service many writes fast by exploiting row buffer locality such that servicing writes next time can be performed a long time later.

We extensively analyze and compare DRAM-aware writeback and eager writeback in Section 6.2.

5. Methodology

5.1. System Model

We use a cycle accurate x86 CMP simulator for our evaluation. Our simulator faithfully models all microarchitectural details such as bank conflicts, port contention, and buffer/queuing delays. The baseline configuration of processing cores and the memory system for single, 4, and 8-core CMP systems is shown in Table 3. Our simulator also models DDR3 DRAM performance-related timing constraints in detail as shown in Table 4. To evaluate the effectiveness of our mechanism in systems with prefetching (discussed in Section 6.4), we employ an aggressive stream prefetcher [19] (32 streams, prefetch degree of 4, prefetch distance of 6 cache lines) for each core.

Execution Core	4.8 GHz, Out of order, 15 (fetch, decode, rename stages) stages, decode/retire up to 4 instructions, issue/execute up to 8 microinstructions; 256-entry reorder buffer; 32-entry load-store queue; 256 physical registers
Front End	Fetch up to 2 branches; 4K-entry BTB; 64-entry return address stack; Hybrid branch predictor: 64K-entry gshare and 64K-entry PAs predictor with 64K-entry selector
Caches and on-chip buffers	L1 I-cache: 32KB, 4-way, 2-cycle, 1 read port, 1 write port, 64B line size; L1 D-cache: 32KB, 4-way, 4-bank, 2-cycle, 1 read port, 1 write port, 64B line size; Shared last-level cache: 8-way, 8-bank, 15-cycle, 1 read/write port per bank, writeback, 64B line size, 1, 2, 4MB for single, 4 and 8-core CMPs; 32, 128, 256-entry MSHRs, 32, 128, 256-entry L2 access/miss/fill buffer for single, 4 and 8-core CMPs
DRAM and bus	1, 2, 2 channels (memory controllers) for 1, 4, 8-core CMPs; 800MHz DRAM bus cycle, Double Data Rate (DDR3 1600MHz) [9]; 8B-wide data bus per channel, BL = 8; 1 rank, 8 banks per channel, 8KB row buffer per bank;
DRAM controllers	On-chip, open-row, FR-FCFS scheduling policy [15]; 64-entry (8×8 banks) DRAM read and write buffers per channel, drain_when_full write buffer policy

Table 3. Baseline configuration

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^tRP	11	Activate to read/write	^tRCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^tRC	39
Activate to precharge	^tRAS	28	Read to precharge	^tRTP	6
Burst length	^tBL	4	Column address strobe to column address strobe	^tCCD	4
Activate to activate (different bank)	^tRRD	6	Four activate windows	^tFAW	24
Write to read	^tWTR	6	Write recovery	^tWR	12

Table 4. DDR3 1600 DRAM timing specifications

5.2. Metrics

To measure multi-core system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [18], and *Harmonic mean of Speedups (HS)* [8]. As shown by Eyerman and Eeckhout [1], WS corresponds to system throughput and HS corresponds to the inverse of job turnaround time. In the equations that follow, N is the number of cores in the CMP system. IPC_i^{alone} is the IPC measured when application i runs alone on one core of the CMP system (other cores are idle, therefore application i can utilize all of the shared resources) and $IPC_i^{together}$ is the IPC measured when application i runs on one core while other applications are running on the other cores of the CMP system.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad WS = \sum_i^N \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad HS = \frac{N}{\sum_i^N \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

5.3. Workloads

We use the SPEC CPU 2000/2006 benchmarks for experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [14] as a representative portion of each benchmark.

We evaluate 18 SPEC benchmarks on the single-core system. The 16 benchmarks (which have at least 10% ideal performance improvement when all writes are removed) discussed in Figure 1 and the two benchmarks, *vpr* and *twolf* mentioned in Section 4.2. The characteristics of the 18 SPEC benchmarks are shown in Table 5. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran 30 and 12 randomly chosen workload combinations for our 4 and 8-core CMP configurations respectively.

Benchmark	Type	IPC	MPKI	RHR	Benchmark	Type	IPC	MPKI	RHR
171.swim	FP00	0.35	23.10	47.84	173.applu	FP00	0.93	11.40	82.12
175.vpr	IN00	1.02	0.89	18.36	178.galgel	FP00	1.42	4.84	46.14
179.art	FP00	0.26	90.92	94.59	189.lucas	FP00	0.61	10.61	56.09
191.fma3d	FP00	1.01	4.13	73.48	300.twolf	INT00	0.98	0.72	31.12
429.mcf	INT06	0.15	33.64	17.93	433.milc	FP06	0.48	29.33	84.39
436.cactusADM	FP06	0.63	4.51	12.94	450.soplex	FP06	0.40	21.24	75.76
459.GemsFDTD	FP06	0.49	15.63	47.28	462.libquantum	INT06	0.67	13.51	93.20
470.lbm	FP06	0.46	20.16	66.58	471.omnetpp	INT06	0.49	10.11	46.93
473.astar	INT06	0.47	10.19	42.62	481.wrf	FP06	0.72	8.11	73.71

Table 5. Characteristics for 18 SPEC benchmarks: IPC, MPKI (last-level cache misses per 1K instructions), DRAM row-hit rate (RHR)

6. Experimental Evaluation

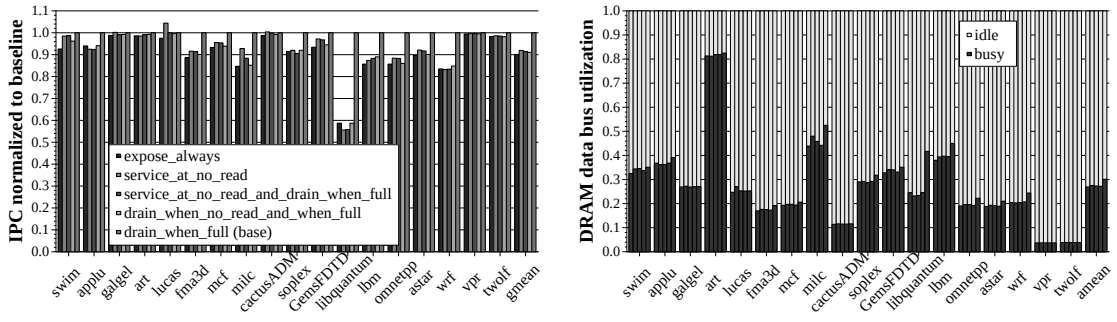
We first show that the baseline write buffer management policy that we use outperforms other policies and then we analyze how our proposed DRAM-aware writeback mechanism works for single and multi-core systems.

6.1. Performance of Write Buffer Management Policies

In addition to our baseline (*drain_when_full*), we evaluate four write buffer management policies that are all based on the same principle as previous work [6, 12, 17]. The first one, *expose_always*, is a policy that always exposes DRAM writes and reads to the DRAM controller together. The DRAM controller makes scheduling decisions based on the baseline FR-FCFS scheduling policy while always prioritizing reads over writes. However, if all DRAM timing constraints are satisfied for a write, the write can be scheduled even though there are reads in the read request buffer. For example, while a precharge for a read is in progress in one bank, a row-hit write in a different bank can be scheduled and serviced if all timing constraints for the write are satisfied (assuming there is no pending read to the corresponding bank). The second policy is *service_at_no_read* which was discussed in Section 1. This policy exposes writes to the DRAM controller only when there is no pending read request or when the write buffer

is full, and stops exposing writes when a read request arrives or when the write buffer is not full any more. The third policy is *service_at_no_read_and_drain_when_full* which is the same as service_at_no_read except that once the write buffer is full, all writes are exposed until the buffer becomes empty. The fourth policy, *drain_when_no_read_and_when_full* is the same as our baseline policy that exposes all writes and drains the buffer every time the write buffer is full, except that it also keeps exposing all writes until the buffer becomes empty even when writes are exposed due to no pending read in the read request buffer. The DRAM controller follows the FR-FCFS policy to schedule reads and exposed writes for all of the above policies.

Figure 7 shows IPC normalized to the baseline and DRAM data bus utilization on a single-core system for the above five write buffer policies. DRAM bus utilization is calculated by dividing the number of cycles the data bus transfers data (both reads and writes) by the number of total execution cycles. Note that since we only change the write buffer policy, the total number of reads and writes does not change significantly among the five policies. Therefore, we can meaningfully compare the DRAM data bus utilization of each policy as shown in Figure 7(b). A large number of busy cycles indicates high DRAM throughput. On the other hand, a larger number of idle cycles indicates more interference among the requests.



(a) Performance (b) Data bus utilization

Figure 7. Performance and DRAM bus utilization for various write buffer policies

Our baseline *drain_when_full* policy outperforms the other four polices significantly for almost all benchmarks. The other policies cause many idle cycles due to frequent read-to-write and write-to-read switching as shown in Figure 7(b). The *expose_always* policy performs worst since writes are always exposed and can be scheduled more freely than other policies by the DRAM controller, hence the most read-to-write and write-to-read penalties. The *service_at_no_read_and_drain_when_full* and *drain_when_no_read_and_when_full* policies also cause many write-to-read switching penalties by allowing some writes to be scheduled when there is no read in the read buffer thereby resulting in many idle cycles.

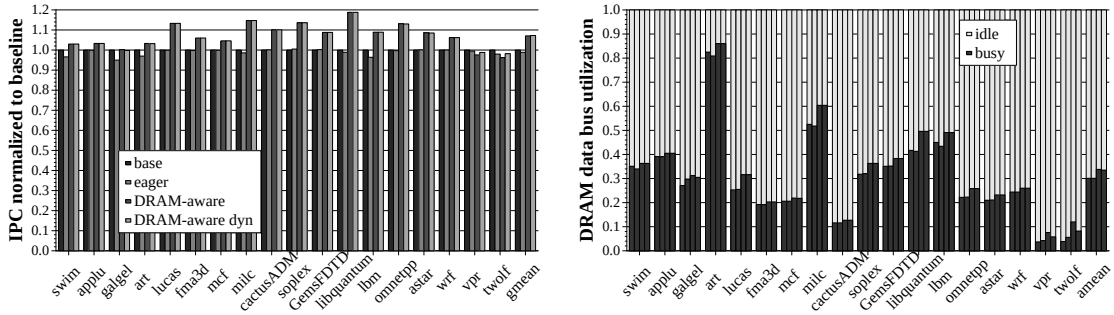
In contrast, the *drain_when_full* policy increases data bus utilization by allowing the DRAM controller to service reads without interference from writes as much as possible. It also reduces write-to-read switching penalties overall because only one write-to-read switching penalty (also one read-to-write penalty) is needed to drain all the writes from the write buffer. Finally it also gives more chances to the DRAM controller to exploit better row buffer locality and DRAM bank-level parallelism (servicing writes to different banks concurrently, if possible) by exposing more writes together. To summarize, the *drain_when_full* policy improves performance by 8.8% on average and increases data bus utilization by 9.4% on average compared to the best of the other four policies.

Note that there is still a significant number of bus idle cycles in Figure 7(b) even with the best policy. Our DRAM-aware writeback mechanism aims to minimize write-caused interference so that idle cycles are better utilized.

6.2. Single-Core Results

This section presents performance evaluation of the DRAM-aware writeback mechanism on a single-core system. Figure 8 shows IPC normalized to the baseline and DRAM data bus utilization for the eager writeback technique, DRAM-aware writeback, and DRAM-aware writeback with the optimization described in Section 4.2. The optimization dynamically adjusts the dirty line LRU

positions which are considered for writeback based on their rewrite rate. When the rewrite rate is less than 50%, we allow any LRU position which generates a row-hit to be written back. If the rewrite rate is between 50% and 90%, only the least recently used half of the LRU stack can be sent out. If the rewrite rate is more than 90%, only writebacks in the LRU position can be sent out. Note that the eager writeback mechanism uses a write buffer policy that sends writes when the bus is idle as pointed out in Section 4.4. In Section 6.1 we showed that sending out writes when the bus is idle is inferior to draining the write buffer only when it is full (as done by *drain_when_full*). As such, for fair comparison we use an improved version of eager writeback that uses the baseline *drain_when_full* policy. First we make the following major performance-related observations from Figure 8 and then provide more insights and supporting data using other DRAM and last-level cache statistics in subsections.



(a) Performance (b) Data bus utilization

Figure 8. Performance and DRAM bus utilization on single-core system

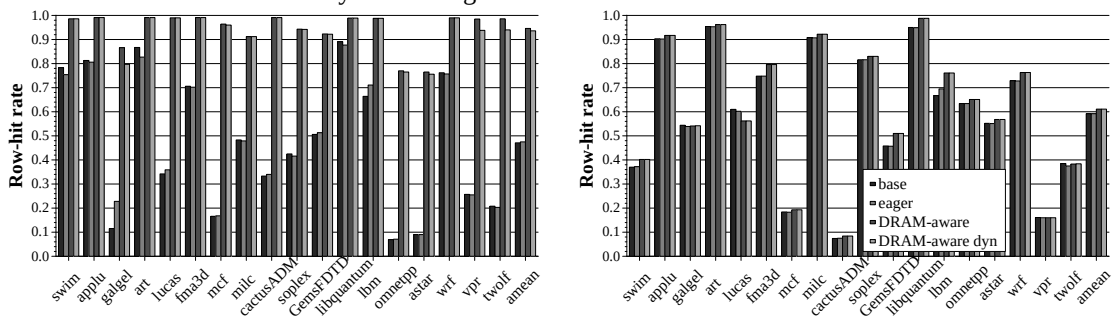
First, the eager writeback technique degrades performance by 1.1% compared to the baseline. This is mainly because it is not aware of DRAM characteristics. Filling the write buffers with writebacks for dirty lines which are in the LRU position of their respective sets does not guarantee fast service time of writes since servicing row-conflict writes must pay the large write-to-precharge penalties. As shown in Figure 8(b), eager writeback suffers as many idle cycles as the baseline on average.

Second, DRAM-aware writeback improves performance for all benchmarks except for *vpr* and *twolf*. It improves performance by more than 10% for *lucas*, *milc*, *cactusADM*, *libquantum* and *omnetpp*. This is because our mechanism sends many row-hit writes that are serviced quickly by the DRAM controller, which in turn reduces write-to-read switching penalties. As shown in Figure 8(b), our mechanism improves DRAM bus utilization by 12.3% on average across all 18 benchmarks. Increased bus utilization translates to high performance. On average, the mechanism improves performance by 7.1%. However, the increased bus utilization does not increase performance for *vpr* and *twolf*. In fact, the mechanism degrades performance for these two benchmarks by 2.4% and 3.8% respectively. This is due to the large number of writebacks that are generated by the DRAM-aware writeback mechanism for these two benchmarks. We developed a dynamic optimization to mitigate this degradation which we refer to as dynamic DRAM-aware writeback.

Dynamic DRAM-aware writeback mitigates the performance degradation for *vpr* and *twolf* by selectively sending writebacks based on the rewrite rate of DRAM-aware writebacks. By doing so, the performance degradation of *vpr* and *twolf* becomes 1.2% and 1.8% respectively, which results in 7.2% average performance improvement for all 18 benchmarks. Note that the dynamic mechanism still achieves almost all of the performance benefits of non-dynamic DRAM-aware writeback for the other 16 benchmarks. As we discussed in Section 4.2, the gain from this optimization is small compared to design effort and hardware cost.

6.2.1. Why Does Eager Writeback Not Perform Well? As discussed above, eager writeback degrades performance compared to the baseline in today's DDR DRAM systems since it generates writebacks in a DRAM-unaware manner. In other words, it can fill the write buffer with many row-conflict writes. Figure 9 shows the row-hit rate for write and read requests serviced by DRAM for the 18 benchmarks. Because we use the open-row policy (that does not use either auto precharge or manual precharge after each

access), row-conflict rate can be calculated by subtracting row-hit rate from one.



(a) Writes (b) Reads

Figure 9. Row-hit rate for DRAM writes and reads

While eager writeback does not change row-hit rates for reads as shown in Figure 9(b), it generates more row-conflict writes (fewer row-hits) for *swim*, *art*, *milc*, and *libquantum* compared to the baseline as shown in Figure 9(a). For these benchmarks, these row-conflict writes introduce many idle cycles during the servicing of writes with the baseline *drain_when_full* write buffer policy as shown in Figure 8(b). This increases the time to drain the write buffer which in turn delays the service of critical reads required for an applications' progress.

6.2.2. How Does DRAM-Aware Writeback Perform Better? In contrast to eager writeback, our mechanism selectively sends many row-hit writes that are serviced quickly by the DRAM controller. Therefore the row-hit rate for writes significantly increases (to 94.6% on average) as shown in Figure 9(a). Note that it also increases the row-hit rate for reads (by 3.3% on average) as shown in Figure 9(b). This is mainly because DRAM-aware writeback reduces row-conflicts between reads and writes as well by reducing write-to-read switching occurrences. We found that due to the last-level cache and row locality of programs, it is very unlikely that while servicing reads to a row, a dirty cache line to that row is evicted from the cache. Therefore decreased write-to-read switching frequency reduces row-conflicts between writes and reads for the entire run of an application.

DRAM-aware writeback leverages the benefits of the write buffer and the *drain_when_full* write buffer policy as discussed in Section 4. Once the mechanism starts sending all possible row-hit writebacks for a row, the write buffer becomes full very quickly. The *drain_when_full* write buffer policy continues to expose writes until the buffer becomes empty. This makes it possible for the DRAM controller to service all possible writes to a row very quickly. Therefore our mechanism reduces the total number of write buffer drains over the entire run of an application. Table 6 provides the evidence of such behavior. It shows the total number of write buffer drains and the average number of writes per write buffer drain for each benchmark. The number of writes per write buffer drain for DRAM-aware writeback is increased significantly compared to the baseline and eager writeback. Therefore the total number of drains is significantly reduced, which indicates that DRAM-aware writeback reduces write-to-read switching frequency thereby increasing row-hit rate for reads. The increased row-hits (i.e., reduced row conflicts) lead to high data bus utilization for both reads and writes and performance improvement as shown in Figure 8.

	Benchmark	swim	applu	galgel	art	lucas	fma3d	mcf	milc	cactusADM	soplex	GemsFDTD	libquantum	lbm	omnetpp	astar	wrf	vpr	twolf
drains	base	64960	24784	2891	83870	19890	24625	62521	50764	15264	43967	49027	115563	92310	35902	26377	38353	1961	4785
	eager	76660	26367	4264	90020	22096	25263	62938	52581	15243	43033	50805	114461	94396	36425	26859	38622	2732	8080
	DRAM-aware	13642	2927	8043	16754	7677	2995	49915	47982	2142	17611	14023	12535	24630	44413	29836	4921	4346	9030
writes/drain	base	25.38	14.36	80.11	23.34	23.93	14.79	34.19	20.43	15.99	17.07	28.21	10.16	22.57	23.22	28.78	13.16	27.54	21.18
	eager	21.52	13.51	97.86	24.29	22.47	14.43	34.09	19.75	16.05	17.53	27.34	10.26	22.19	23.24	28.48	13.08	29.72	27.15
	DRAM-aware	121.90	121.97	50.19	128.26	96.24	122.09	45.05	21.83	114.27	44.32	99.49	93.66	85.08	20.50	27.05	103.26	69.91	71.88

Table 6. Number of write buffer drains and number of writes per drain

6.2.3. When is Dynamic DRAM-Aware Writeback Required? Recall that DRAM-aware writeback degrades performance for *vpr* and *twolf*. Figure 10 shows the total number of DRAM read and write requests serviced by DRAM for the 18 benchmarks. While DRAM-aware writeback does not increase the total number of reads and writes significantly for the other 16 benchmarks like the baseline and eager writeback do, it does increases the number of writes significantly for *vpr* and *twolf*.

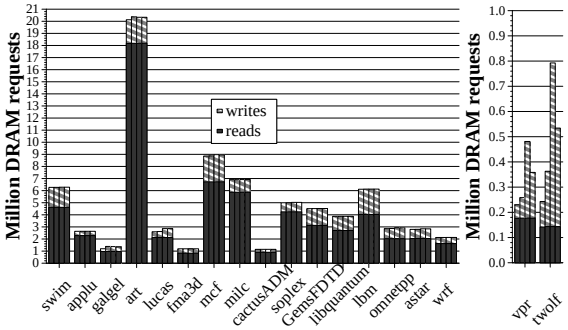


Figure 10. Number of DRAM requests

Table 7 shows the total number of writebacks generated by DRAM-aware writeback, cache lines that were cleaned but reread, and cache lines that were cleaned but rewritten. It also shows the number of rewrites per cache line written back (referred to as rewrite rate). For *vpr* and *twolf*, rewrites to cache lines cleaned by the mechanism happen very frequently (82% and 85% respectively). These rewritten lines' writebacks are sent again by the mechanism thereby increasing the number of writes significantly. Increased writes make the write buffer full frequently, therefore aggregate write-to-read switching penalty becomes larger which degrades performance. However, the performance degradation is not significant because the total number of requests is not large (i.e., memory non-intensive) as shown in Figure 10. .

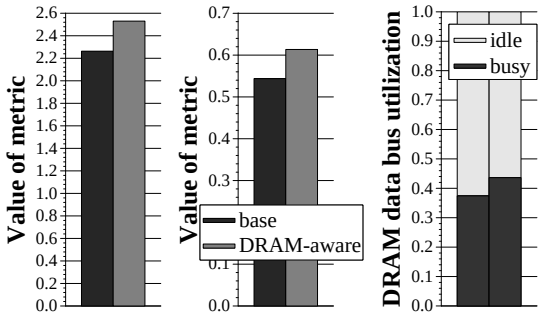
Benchmark	swim	applu	galgel	art	lucas	fma3d	mcf	milc	cactusADM	soplex	GemsFDTD	libquantum	lbm	omnetpp	astar	wrf	vpr	twolf
Writebacks	1640260	350641	346550	2061007	731063	361590	2167616	947328	242377	732556	1251832	1161287	2069208	698896	612423	500963	299262	639582
Reread	42	183	23741	70931	0	0	122290	0	16	1599	1905	0	0	21982	6012	746	12479	24230
Rewritten	20	0	166871	191596	0	501	108871	0	55	28593	13474	0	0	73667	37075	2588	245645	540604
Rewrite Rate	0.00	0.00	0.48	0.09	0.00	0.00	0.05	0.00	0.00	0.04	0.01	0.00	0.00	0.11	0.06	0.01	0.82	0.85

Table 7. Number of DRAM-aware writebacks generated, reread cache lines and rewritten cache lines, and rewrite rate

The dynamic DRAM-aware writeback mechanism discussed in Section 4.2 mitigates this problem by adaptively limiting write-backs based on rewrite rate estimation. Since the rewrite rate is high most of the time for *vpr* and *twolf*, the dynamic mechanism allows writebacks only for row-hit dirty lines which are in the LRU position of their respective sets. Therefore, it reduces the number of writebacks as shown in Figure 10. In this way, it mitigates the performance degradation for these two benchmarks as shown in Figure 8. Note that the dynamic mechanism does not change the benefits of DRAM-aware writeback for the other 16 benchmarks since it adapts itself to the rewrite behavior of the applications.

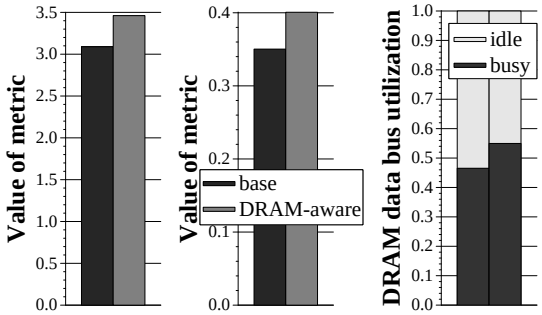
6.3. Multi-Core Results

We also evaluate the DRAM-aware writeback mechanism on multi-core systems. Figures 11 and 12 show average system performance and bus utilization for the 4 and 8-core systems described in Section 5. In multi-core systems, write-caused interference is more severe since there is greater contention between reads and writes in the DRAM system. Furthermore, writes can delay critical reads of all cores. As such, reducing write-caused interference is even more important in multi-core systems. Our DRAM-aware writeback mechanism increases bus utilization by 16.5% and 18.1% for the 4 and 8-core systems respectively. This leads to an increase in weighted speedup (WS) and harmonic speedup (HS) by 11.8% and 12.8% for the 4-core system and by 12.0% and 14.4% for the 8-core system. We conclude that DRAM-aware writeback is effective for multi-core systems.



(a) WS (b) HS (c) Bus utilization

Figure 11. Performance for 30 4-core workloads

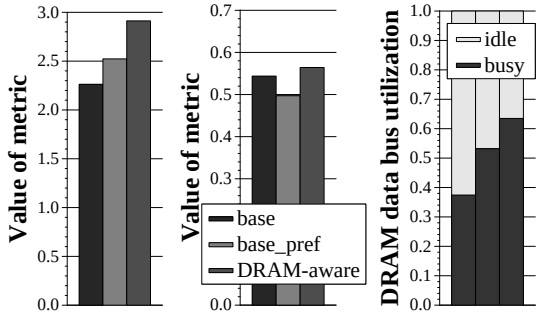


(a) WS (b) HS (c) Bus utilization

Figure 12. Performance for 12 8-core workloads

6.4. Effect on Systems with Prefetching

We evaluate our mechanism when it is employed in a 4-core system with the stream prefetcher described in Section 5. Figure 13 shows average system performance and bus utilization for the baseline with no prefetching, the baseline with prefetching, and the baseline with prefetching and DRAM-aware writeback for our 30 4-core workloads.



(a) WS (b) HS (c) Bus utilization

Figure 13. Performance for 30 4-core workloads when prefetching is enabled

Prefetching increases write-caused interference severely. Prefetch requests, that are essentially reads, put more pressure on the DRAM system. Prefetching improves weighted speedup by 11.5% by utilizing idle DRAM bus cycles while it degrades harmonic speedup by 8.6% compared to the baseline with no prefetching. Using DRAM-aware writeback significantly improves DRAM bus utilization (by 19.4% compared to prefetching) by reducing write-caused interference. The increased bus utilization translates into higher performance. Using DRAM-aware writeback improves WS and HS by 15.4% and 13.5% compared to prefetching alone. We conclude that DRAM-aware writeback is also effective in multi-core systems that employ prefetching.

7. Related Work

7.1. DRAM Access Scheduling and Write Buffer Policies

Many DRAM scheduling policies [21, 15, 12, 13, 10, 11, 2, 5] have been proposed in the literature. Many of them [15, 11, 2, 5] do not address how to manage write-caused interference for high DRAM throughput. In contrast, our mechanism sends writes intelligently from the last-level cache so that overall write-caused interference can be reduced. As such, our techniques are orthogonal to these DRAM scheduling policies. As shown in Section 6.2, our mechanism allows the underlying DRAM controller to better exploit row buffer locality for not only writes but also reads by reducing write-to-read switching penalties.

Other proposals [6, 12, 17] discuss writeback management polices and DRAM scheduling for writes. Their policies are based on the principle that scheduling writes when the bus is idle (no pending reads) can reduce the contention between reads and writes. However, we found that this principle does not work with today's high-bandwidth DDR DRAM systems with their large write-caused latency penalties as shown in Section 6.1. We have shown that servicing all writes when the write buffer is full is the best since it reduces write-to-read switching penalties and allows the DRAM controller to better exploit row buffer locality and bank-level parallelism exposed by more writes.

7.2. Last-Level Cache Management

Many cache replacement and insertion policies have been proposed. These are all orthogonal to our work since our mechanism does not change the underlying replacement or insertion policies. If a writeback of a dirty line is sent by our mechanism, the dirty line becomes non-dirty and stays in the cache. When any replacement policy decides to evict such a clean cache line, the line is simply removed.

A number of prior papers propose aggressive early writeback policies [6, 7, 4, 20] which send writebacks of dirty lines before they are evicted by a replacement policy. We have already compared our mechanism to eager writeback [6] both qualitatively and quantitatively in Sections 4.4 and 6.2. Other proposals [7, 4, 20] periodically send early writebacks to the next-level cache or DRAM to increase the reliability of on-chip caches with low cost. Even though the motivation for our mechanism is not to improve reliability, DRAM-aware writeback can help reduce vulnerability in the last-level cache since it aggressively sends writebacks just like other early writeback policies do.

8. Conclusion

This paper described the problem of write-caused interference in the DRAM system, and showed it has significant performance impact in modern processors. Write-caused interference will continue to be a performance bottleneck in the future because the memory systems operating frequency continues to increase in order to provide more memory bandwidth. To reduce write-caused interference, we proposed a new writeback policy for the last-level cache, called DRAM-aware writeback, which aggressively sends out writebacks for dirty lines that can be quickly written back to DRAM by exploiting row buffer locality. We demonstrated that the proposed mechanism and the previous best write buffer management policy are synergistic in that they work together to reduce write-caused interference by allowing the DRAM controller to service many writes quickly together. This reduces the delays incurred by read requests and therefore increases performance significantly in both single-core and multi-core systems. We also showed that the performance benefits of the mechanism increases in multi-core systems or systems with prefetching where there is higher contention between reads and writes in the DRAM system. We conclude that DRAM-aware writeback can be a simple solution to reduce write-caused interference.

Our mechanism is not limited to DRAM technology since other memory technologies also suffer from the write-caused interference problem. A high data transfer frequency in the memory system makes write-to-read and even write-to-write latencies in each

bank very costly just as in today's high bandwidth DDR DRAM systems. An avenue of future work is to reduce the write-back interference in other, emerging memory technologies.

References

[1] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.
[2] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA-35*, 2008.
[3] JEDEC. *JEDEC Standard: DDR3 SDRAM STANDARD (JESD79-3D)*. http://www.jedec.org/standards-documents/docs/jesd-79-3d.
[4] S. Kim. Area-efficient error protection for caches. In *DATE*, 2006.
[5] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.
[6] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *MICRO-33*, 2000.
[7] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: A data cache perspective. In *ISLPED*, 2004.
[8] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001.
[9] Micron. *2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*. http://download.micron.com/pdf/datasheets/dram/ddr3/.
[10] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
[11] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.
[12] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, pages 80–87, 2004.
[13] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[14] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[15] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[16] Samsung. *Application Note: tWR (Write Recovery Time)*. http://www.samsung.com/global/business/semiconductor/products/dram/downloads/.
[17] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.
[18] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-9*, pages 164–171, 2000.
[19] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.
[20] D. H. Yoon and M. Erez. Memory mapped ECC: low-cost error protection for last level caches. In *ISCA-36*, 2009.
[21] W. Zuravleff and T. Robinbson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. Patent Number 5,630,096, 1997.